Logical Methods in Computer Science
Vol. 6 (3:15) 2010, pp. 1–1–23
www.lmcs-online.org

Submitted Nov. 13, 2009
Published Sep. 3, 2010

INITIAL ALGEBRA SEMANTICS FOR CYCLIC SHARING TREE STRUCTURES [*]

MAKOTO HAMANA

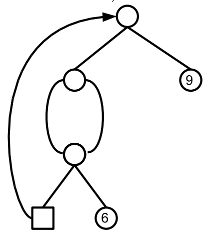
Department of Computer Science, Gunma University, Japan
e-mail address: hamana@cs.gunma-u.ac.jp

Abstract. Terms are a concise representation of tree structures. Since they can be naturally defined by an inductive type, they offer data structures in functional programming and mechanised reasoning with useful principles such as structural induction and structural recursion. However, for graphs or "tree-like" structures – trees involving cycles and sharing – it remains unclear what kind of inductive structures exists and how we can faithfully assign a term representation of them. In this paper we propose a simple term syntax for cyclic sharing structures that admits structural induction and recursion principles. We show that the obtained syntax is directly usable in the functional language Haskell and the proof assistant Agda, as well as ordinary data structures such as lists and trees. To achieve this goal, we use a categorical approach to initial algebra semantics in a presheaf category. That approach follows the line of Fiore, Plotkin and Turi's models of abstract syntax with variable binding.

1. Introduction

Terms are a convenient, concise and mathematically clean representation of tree structures used in logic and theoretical computer science. In the fields of traditional algorithms and graph theory, one usually uses unstructured representations for trees, such as pairs (V, E) of vertices and edges sets, adjacency lists, adjacency matrices, pointer structures, etc. Such representations are more complex and unreadable than terms. We know that term representation provides a well-structured, compact and more readable notation.

However, consider the case of a "tree-like" structure such as that depicted below.



This kind of structure – graphs, but almost trees involving (a few) exceptional edges – quite often appears in logic and computer science. Examples include internal representations of expressions in implementations of functional languages that share common sub-expressions for efficiency, data models of XML such as trees with pointers [CGZ05], proof trees admitting cycles for cyclic proofs [Bro05], term graphs in graph rewriting [BvEG+87, AK96], and control flow graphs of imperative programs used in static

2000 ACM Subject Classification: D.3.2, E.1, F.3.2.
Key words and phrases: graphs, term graphs, dependent types, Haskell, variable binding.
[*] Some of the results presented here were first published in TLCA proceedings [Ham09].

analysis and compiler optimizations [CFR$^+$91]. Suppose that we want to treat such structures in a pure functional programming language such as Haskell, Clean, or a proof assistant such as Coq, Agda [Nor07]. In such a case, we would have to abandon the use of naive term representation, and would instead be compelled to use an unstructured representation such as (V, E), adjacency lists, etc. Furthermore, a serious problem is that we would have to abandon structural recursion and induction to decompose them because they look "tree-like" but are in fact graphs, so there is no obvious inductive structure in them. This means that in functional programming, we cannot use pattern matching to treat tree-like structures, which greatly decreases their convenience. This lack of structural induction implies a failure of being an inductive type. But, are there really no inductive structures in tree-like structures? As might be readily apparent, tree-like structures are almost trees and merely contain finite pieces of information. The only differences are the presence of "cycles" and "sharing".

In this paper, we give an initial algebra characterisation of cyclic sharing tree structures in the framework of categorical universal algebra. The aim of this paper is to derive the following practical goals *from the initial algebra characterisation*.

[I] To develop a simple term syntax for cyclic sharing structures that admits structural induction and structural recursion principles.

[II] To make the obtained syntax directly usable in the current functional languages and proof assistants, as well as ordinary data structures, such as lists and trees.

The goal [I] requires that the term syntax *exactly* represents cyclic sharing structures (i.e. no junk terms exist) to make structural induction possible. The goal [II] requires that the obtained syntax should be *lightweight as possible*, which means that e.g. well-formedness and equality tests on terms for cyclic sharing structures should be fast and easy, as are ordinary data structures such as lists and trees. We do not want many axioms to characterise the intended structures, because, in programming situation, checking the validity of axioms every time is expensive and makes everything complicated. Ideally, formulating structures *without axioms* is best. Therefore, the goal [II] is rephrased more specifically as:

[II'] To give an inductive type that represents cyclic sharing structures *uniquely*. We therefore rely on that a type checker automatically ensures the well-formedness of cyclic sharing structures.

To show this, we give concrete definitions of types for cyclic sharing structures in two systems: a functional programming language Haskell and a proof assistant Agda.

1.1. **Variations on initial algebra semantics.**

The initial algebra semantics models syntax/datatype as the initial algebra and semantics as another algebra, and the compositional interpretation by the unique homomorphism. The classical formulation of initial algebra semantics for syntax/datatype taken by ADJ [GTW76] is categorically reformulated as an initial algebra of a functor in the category **Set** of sets and functions [Rob02], which means that carriers are merely sets and operations are functions.

Recently, varying the base category other than **Set**, initial algebra semantics for algebras of functors has proved to be useful framework for characterisation of various mathematical and computational structures in a uniform setting. We list several: S-sorted abstract syntax is characterised as initial algebra in $\mathbf{Set}^S$ [Rob02], second-order abstract syntax as initial algebra in $\mathbf{Set}^{\mathbb{F}}$ [FPT99, Ham04, Fio08] (where $\mathbb{F}$ is the category of finite sets), explicit substitutions as initial algebras in the category $[\mathbf{Set}, \mathbf{Set}]_f$ of finitary functors [GUH06], recursive path ordering for term rewriting systems as algebras in the category **LO** of linear orders [Has02], second-order rewriting systems as initial algebras in the preorder-valued functor category $\mathbf{Pre}^{\mathbb{F}}$ [Ham05], and nested datatypes [GJ07] and generalised algebraic datatypes (GADTs) [JG08] in functional programming as initial algebras in $[\mathcal{C},\mathcal{C}]$ and $[|\mathcal{C}|,\mathcal{C}]$, respectively, where $\mathcal{C}$ is a ω-cocomplete category.

This paper adds a further example to the list given above. We characterise cyclic sharing structures as an initial algebra in the category $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$, where $\mathbb{T}$ is the set of all "shapes" of trees and $\mathbb{T}^*$ is the set of all tree shape contexts. We derive structural induction and recursion principles from it. An important point is that we merely use *algebra of functor* to formulate cyclic sharing structures, i.e. not (models of) equational specifications or (Σ, E)-algebras. This characterisation achieves the requirement of "without axioms". Moreover, it is the key to formulate them by an inductive type.

1.2. **Basic idea.** It is known in the field of graph algorithms [Tar72] that, by traversing a rooted directed graph in a depth-first search manner, we obtain a *depth-first search tree*, which consists of a spanning tree (whose edges are called *tree edges*) of the graph and *forward edges* (which connect ancestors in the tree to descendants), *back edges* (the reverse), and *cross edges* (which connect nodes across the tree from right to left).

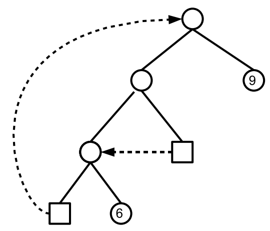


Figure 1: Depth-first search tree

Forward edges can be decomposed into tree and cross edges by placing indirect nodes. For example, the graph in the front page becomes a depth-first search tree in Fig.1 where solid lines are tree edges and dashed lines are back and cross edges. This is the target structure we will model in this paper. That is, tree edges are the basis of an *inductive structure*, back edges are used to form *cycles*, and cross edges are used to form *sharing*. Consequently, our task is to seek how to characterise pointers that make back edges and cross edges in inductive constructions.

1.3. **Formulation.** The crucial idea to formulate pointers in inductive constructions is to use *binders* as *pointers* in abstract syntax. Trees are formulated as terms. Hence, a remaining problem is how to exactly capture binders in terms. Fiore, Plotkin and Turi [FPT99] have characterised abstract syntax with variable binding by initial algebras in the presheaf category $\mathbf{Set}^{\mathbb{F}}$, where $\mathbb{F}$ is the category of finite sets. For example, abstract syntax of λ-terms is modeled as a functor

$$\Lambda : \mathbb{F} \longrightarrow \mathbf{Set}$$

equipped with three constructors for λ-terms as an algebra structure on Λ. Each set $\Lambda(X)$ gives the set of all λ-terms which may contain free variables taken from a set X in $\mathbb{F}$. This formulation models a structure (here, abstract syntax trees) indexed by suitable invariant (here, free variables considered as contexts), which is essential information to capture the intended structure (abstract syntax with variable binding).

However, this approach using algebras in $\mathbf{Set}^{\mathbb{F}}$ is insufficient to represent "cross edges" in tree-like graphs. Ariola and Klop [AK96] have analysed that there are two kinds of sharing in this kind of tree-like graphs:

(i) vertical sharing (i.e. back edges in depth-first search trees), and

(ii) horizontal sharing (i.e. cross edges).

In principle, binders capture "vertical" contexts only, but to represent cross edges exactly, we must capture "horizontal" context information that cannot be handled by the index category $\mathbb{F}$.

To solve this problem, in this paper we take a richer index category that is enough to model cross edges. We introduce the notion of *shape trees* and contexts consisting of them, which represents other parts of tree viewing from a pointer node. We use the set $\mathbb{T}$ of all shape trees as "types" of syntax, and the set $\mathbb{T}^*$ of all sequences of shape trees as "context". We follow Fiore's treatment of initial algebra semantics for typed abstract syntax with variable binding [Fio02] by algebras in the presheaf category $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Therefore, cyclic sharing trees are modelled as a $\mathbb{T}$ and $\mathbb{T}^*$-indexed set

$$T : \mathbb{T} \longrightarrow (\mathbb{T}^* \longrightarrow \mathbf{Set})$$

equipped with constructors of cyclic sharing trees as an algebra structure.

1.4. **Organisation.** We first give types and abstract syntax for cyclic sharing binary trees in Section 2.2. We then characterise cyclic sharing binary trees as an initial algebra in Section 3. Section 4 gives a way of implementing cyclic sharing structures by inductive types. Section 5 generalises our treatment to arbitrary signature for cyclic sharing structures. Section 6 presents discussion of variations of the form of pointers in cyclic sharing trees. Section 7 relates our representation and equational term graphs in the initial algebra framework by giving a homomorphic translation. In Section 8, we discuss connections to other approaches to cyclic sharing structures.

2. Abstract Syntax for Cyclic Sharing Structures

2.1. **Cyclic structures by μ-terms.** The μ-notation (μ-terms) for fixed point expressions is widely used in computer science and logic. Its theory has been investigated thoroughly, for example, in [AK96, SP00]. The language of μ-terms suffices to express all cyclic structures.

For example, a cyclic binary tree shown in Fig. 2 (i) is representable as the term

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\ x,\ \mathsf{lf}(7)\ )) \tag{2.1}$$

where bin and lf respectively denote a binary node and a leaf. The point is that the variable x refers to the root labeled by a μ-binder, hence a cycle is represented. To uniquely formulate cyclic structures, here we introduce the following assumption: we attach μ-binders in front of bin only, and put exactly one μ-binder for each occurrence of bin as for (2.1). This is seen as uniform addressing of bin-node, i.e., x, y_1, y_2 are seen as labels or "addresses" of bin-nodes. We also assume no axiom to equate μ-terms. That is, we do not identify a μ-term with its unfolding, since they are different (shapes of) graphs. In summary, μ-terms represent cyclic structures. This is the underlying idea of a representation of cyclic data given in [GHUV06] by using the functional programming language Haskell.

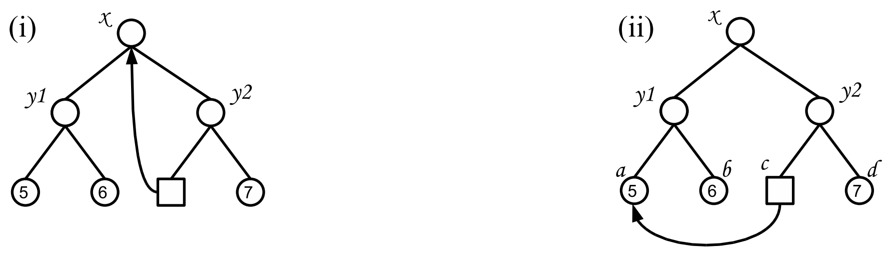


Figure 2: Trees involving cycle and sharing

2.2. **How to represent sharing.** Next, we incorporate sharing. The presence of sharing makes the situation more difficult. Consider the tree (ii) in Fig. 2 involving sharing via a cross edge. As similar to the case of cycles, this might be written as a μ-term

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\boxed{}, \mathsf{lf}(7))).$$

But can we fill the blank to refer the node a (in Fig. 2 (ii)) from the node c "horizontally" by using the mechanism of binders? Actually, μ-binders are insufficient for this purpose. Therefore, we introduce a new notation "$\swarrow p \uparrow x$" to refer to a node horizontally. This notation means going up to a node x labelled by a μ-binder and going down to a position p in the subtree rooted by the node x. In the example presented above, the blank is filled as

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\swarrow 11 \uparrow x\, , \mathsf{lf}(7))).$$

The pointer $\swarrow 11 \uparrow x$ means going back to the node x, then going down through the left child twice (using the position 11). See also Example 2.1. In this section, we focus on the formulation of binary trees involving cycles and sharing. Binary trees are the minimal case that can involve the notion of sharing in structures. Later, in Section 5, we will consider general data types.

2.3. **Shape trees.** We designate our target data structures as *cyclic sharing trees* and its syntactic representation *cyclic sharing terms*. Cyclic sharing trees are binary trees generated by nodes of three kinds, i.e., pointer, leaf, and binary node, and satisfying a certain condition of well-formedness.

To ensure correct sharing, we introduce the notion of *shape trees*, which are skeletons of cyclic sharing trees. That is, shape trees are binary trees, forgetting values in pointer nodes and leaves from cyclic sharing trees. The set $\mathbb{T}$ of all shape trees is defined by

$$\mathbb{T} \ni \tau ::= \mathrm{E} \mid \mathrm{P} \mid \mathrm{L} \mid \mathrm{B}(\tau_1, \tau_2)$$

where E is the void shape, P is the pointer node shape, L is the leaf node shape, and $\mathrm{B}(\tau_1, \tau_2)$ is the binary node shape. We typically use Greek letters σ, τ to denote shape trees.

We define referable positions in a shape tree. A *position* is a finite sequence of $\{1, 2\}$. The root position is denoted by the empty sequence ϵ and the concatenation of positions is denoted by pq or $p.q$. The set $\mathcal{P}os(\tau)$ of referable positions in a shape tree τ is defined by

$$\begin{aligned}
\mathcal{P}os(\mathrm{E}) &= \mathcal{P}os(\mathrm{P}) = \varnothing \\
\mathcal{P}os(\mathrm{L}) &= \{\epsilon\} \\
\mathcal{P}os(\mathrm{B}(\sigma, \tau)) &= \{\epsilon\} \cup \{1p \mid p \in \mathcal{P}os(\sigma)\} \cup \{2p \mid p \in \mathcal{P}os(\tau)\}.
\end{aligned}$$

An important point is that the void E and the pointer P nodes are not referable by other nodes, hence their positions are defined to be empty sets.

2.4. **Syntax and types.** Shape trees are used as types in a typing judgment. As usual, a typing context Γ is a sequence of (variable, shape tree)-pairs.

Typing rules

$$\text{(Pointer)}\frac{p \in \mathcal{P}os(\sigma)}{\Gamma, x:\sigma, \Gamma' \vdash \swarrow p{\uparrow}x : \text{P}} \qquad \text{(Leaf)}\frac{k \in \mathbb{Z}}{\Gamma \vdash \mathsf{lf}(k) : \text{L}}$$

$$\text{(Node)}\frac{x:\text{B}(\text{E},\text{E}), \Gamma \vdash s:\sigma \quad x:\text{B}(\sigma,\text{E}), \Gamma \vdash t:\tau}{\Gamma \vdash \mu x.\mathsf{bin}(s,t) : \text{B}(\sigma,\tau)}$$

In these typing rules, a shape tree type is assigned to the corresponding tree node. That is, a binary node is of type $\text{B}(\sigma,\tau)$ of binary node shape, a pointer node is of type P of pointer node shape, and a leaf node is of type L of leaf node shape.

A type declaration $x:\sigma$ in a typing context (roughly) means that σ is the shape of subtree (say, t) headed by a binder μx (see Example 2.1). Hence, in (Pointer) rule, taking a position $p \in \mathcal{P}os(\sigma)$, we safely refer to a position in the tree t. The notation $\swarrow p{\uparrow}x$ is designed to realise a right-to-left cross edge. Note also that a path obtained by $\swarrow p{\uparrow}x$ is the shortest path from the pointer node to the node referred by $\swarrow p{\uparrow}x$. When $p = \epsilon$, we abbreviate $\swarrow \epsilon{\uparrow}x$ as ${\uparrow}x$. This ${\uparrow}x$ exactly expresses a back edge. In (Node) rule, the shape trees $\text{B}(\text{E},\text{E})$ and $\text{B}(\sigma,\text{E})$ mask nodes that are reachable via left-to-right references (i.e. not our requirement) or redundant references (e.g. going up to a node x then going back down through the same path) by the void shape E.

Example 2.1. The binary tree involving sharing in Fig. 2 (ii) is represented as a well-typed term

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5),\mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\swarrow 11{\uparrow}x, \mathsf{lf}(7))).$$

Its typing derivation is the following.

$$\dfrac{\dfrac{y_1{:}\alpha, x{:}\alpha \vdash \mathsf{lf}(5):\text{L} \quad y_1{:}\text{B}(\text{L},\text{E}), x{:}\alpha \vdash \mathsf{lf}(6):\text{L}}{x{:}\alpha \vdash \mu y_1.\mathsf{bin}(\mathsf{lf}(5),\mathsf{lf}(6)):\text{B}(\text{L},\text{L})} \quad \dfrac{\dfrac{11 \in \mathcal{P}os(\beta)}{y_2{:}\alpha, x{:}\beta \vdash \swarrow 11{\uparrow}x:\text{P}} \quad y_2{:}\text{B}(\text{P},\text{E}), x{:}\beta \vdash \mathsf{lf}(7):\text{L}}{x{:}\beta \vdash \mu y_2.\mathsf{bin}(\swarrow 11{\uparrow}x, \mathsf{lf}(7)):\text{B}(\text{P},\text{L})}}{\vdash \mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5,\mathsf{lf}(6)), \mu y_2.\mathsf{bin}(\swarrow 11{\uparrow}x, \mathsf{lf}(7))):\text{B}(\text{B}(\text{L},\text{L}),\text{B}(\text{P},\text{L}))}$$

where $\alpha = \text{B}(\text{E},\text{E}),\ \beta = \text{B}(\text{B}(\text{L},\text{L}),\text{E})$.

As a result, we can ensure that no dangling pointer happens in this type system.

Theorem 2.2 (Safety)**.** *If a closed term $\vdash t:\tau$ is derivable, any pointer in t points to a node in t.*

Proof. By the typing rules, it is obvious that a variable x in a pointer in the resulting t is always taken from a μ-binder in t. Looking at (Node) rule from the lower to the upper, the shape $\text{B}(\sigma,\tau)$ is always decomposed, and two shape trees in typing contexts at the upper contain fewer possible positions than the lower, i.e.

$$\mathcal{P}os(\text{B}(\text{E},\text{E})) \subseteq \mathcal{P}os(\text{B}(\sigma,\text{E})) \subseteq \mathcal{P}os(\text{B}(\sigma,\tau)).$$

This means that at any application of (Pointer) rule at the topmost of a typing derivation tree, a taken position p is included in the positions of a sub-shape tree of the type of t at the bottom of the typing derivation. $\square$

2.5. **De Bruijn version.** Instead of named variables for binders, a de Bruijn notation is also possible. The construction rules are reformulated as follows. Now a typing context Γ is simply a sequence of shape trees $\tau_1, \ldots, \tau_n$. Let $|\Gamma|$ denote its length. A judgment $\Gamma \vdash t : \tau$ denotes a well-formed term t of shape τ containing free variables (de Bruijn indices) from 1 to $|\Gamma|$. The intended meaning is that the length $|\Gamma|$ denotes how many maximally we can go up from the current node t, and each shape tree τ_i in Γ denotes the shape of the subtree at i-th upped node from t. Consequently, when t is a pointer, a context specifies the set of all positions to which a pointer node can refer.

As known from λ-calculus, using de Bruijn notation, binders become nameless. Therefore we can safely omit "x" from μx. Because the typing rules are designed to attach exactly one μ-binder for each bin, even "μ" can be omitted. As a result, we obtain a simplified construction rules of terms.

Typing rules (de Bruijn version)

$$(\text{dbPointer})\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma, \sigma, \Gamma' \vdash \swarrow p{\uparrow}i : \mathrm{P}} \qquad (\text{dbLeaf})\frac{k \in \mathbb{Z}}{\Gamma \vdash \mathsf{lf}(k) : \mathrm{L}}$$

$$(\text{dbNode})\frac{\mathrm{B}(\mathrm{E},\mathrm{E}), \Gamma \vdash s : \sigma \quad \mathrm{B}(\sigma, \mathrm{E}), \Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{bin}(s,t) : \mathrm{B}(\sigma,\tau)}$$

In the (dbPointer) rule, the condition $|\Gamma| = i - 1$ states that the shape tree σ appears at i-th position of the typing context in the lower judgment. Because its depth-first search tree is unique for a given graph, the following is immediate.

Theorem 2.3 (Uniqueness)**.** *Given a rooted graph that is connected, directed and edge-ordered with each node having out-degree at most* 2*, the term representation in de Bruijn is unique.*

Remark 2.4. This uniqueness of term representation has practical importance. For instance, for the graph in the tree (ii) in Fig. 2, there is only one way to represent it in this term syntax, i.e., $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5, \mathsf{lf}(6)), \mathsf{bin}(\swarrow 11{\uparrow}2, \mathsf{lf}(7)))$ in de Bruijn. Therefore, we do not need any complex equality on graphs (other than the syntactic equality) to check whether given data are the required data. This contrasts directly to other approaches. If we represent a graph as a term graph with labels [BvEG$^+$87], an equational term graph [AK96], or a **letrec**-term [Has97], then several syntactic representations exist for a single graph. Therefore, some normalisation is required, for instance when defining a function on graphs. Generally speaking, our terms are regarded as "de Bruijn notation" of term graphs with labels [BvEG$^+$87].

3. Initial Algebra Semantics

In this section, we show that cyclic sharing terms form an initial algebra and derive structural recursion and induction from it.

3.1. **Construction.** We use Fiore's approach to algebras for typed abstract syntax with binding [Fio02] in the presheaf category $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ where U is the set of all types. Now, we take the set $\mathbb{T}$ of all shape trees for U, and the set $\mathbb{N}$ of natural numbers for variables (i.e. pointers), instead of the category $\mathbb{F}$ of finite sets and all functions (used for renaming variables), because we do not need renaming of pointers.

We define the discrete category $\mathbb{T}^*$ by taking contexts $\Gamma = \langle \tau_1, \ldots, \tau_n \rangle$ as objects (which is equivalent to $\mathbb{N}\downarrow\mathbb{T}$). We also regard $\mathbb{T}$ as a discrete category. We consider algebras in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Two preliminary definitions are required. We define the presheaf $\mathrm{PO} \in \mathbf{Set}^{\mathbb{T}^*}$ for pointers by

$$\mathrm{PO}(\langle \tau_1, \ldots, \tau_n \rangle) = \{\swarrow p \uparrow i \mid 1 \le i \le n,\ p \in \mathcal{P}os(\tau_i)\}.$$

For each $\tau \in \mathbb{T}$, we define the functor $\delta_\tau : \mathbf{Set}^{\mathbb{T}^*} \longrightarrow \mathbf{Set}^{\mathbb{T}^*}$ for context extension by $\delta_\tau A = A(\langle \tau, - \rangle)$.

We define the *signature functor* $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \longrightarrow (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ *for cyclic sharing binary trees*, which takes $A \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ and a type in $\mathbb{T}$, and gives a presheaf in $\mathbf{Set}^{\mathbb{T}^*}$, as follows:

$$(\Sigma A)_{\mathrm{E}} = 0 \qquad (\Sigma A)_{\mathrm{P}} = \mathrm{PO} \qquad (\Sigma A)_{\mathrm{L}} = K_{\mathbb{Z}} \qquad (\Sigma A)_{\mathrm{B}(\sigma,\tau)} = \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})} A_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})} A_\tau$$

where $K_{\mathbb{Z}}$ is the constant functor to $\mathbb{Z}$, and 0 is the empty set functor. A Σ-*algebra* A is a pair (A, α) consisting of a presheaf $A \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ for a carrier and a natural transformation $\alpha : \Sigma A \to A$ for an algebra structure. By definition of Σ, to give an algebra structure is to give the following morphisms of $\mathbf{Set}^{\mathbb{T}^*}$:

$$\mathsf{ptr}^A : \mathrm{PO} \to A_{\mathrm{P}} \qquad \mathsf{lf}^A : K_{\mathbb{Z}} \to A_{\mathrm{L}} \qquad \mathsf{bin}^{\sigma,\tau\, A} : \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})} A_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})} A_\tau \to A_{\mathrm{B}(\sigma,\tau)}.$$

A *homomorphism* ϕ of Σ-algebras from (A, α) to (B, β) is a morphism $\phi : A \to B$ such that $\phi \circ \alpha = \beta \circ \Sigma\phi$.

Let T be the presheaf of all derivable cyclic sharing terms defined by

$$T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}.$$

Theorem 3.1. *For the signature functor Σ for cyclic sharing binary trees, T forms an initial Σ-algebra.*

Proof. Since δ_τ preserves ω-colimits, so does Σ. An initial Σ-algebra is constructed by the colimit of the ω-chain $0 \to \Sigma 0 \to \Sigma^2 0 \to \cdots$ [SP82]. These construction steps correspond to derivations of terms by typing rules, hence their union T is the colimit. The algebra structure $\mathsf{in} : \Sigma T \to T$ of the initial algebra is obtained by one-step inference of the typing rules, i.e., given by the following operations

$$\begin{array}{rll} \mathsf{ptr}^T(\Gamma) : \mathrm{PO}(\Gamma) & \to T_{\mathrm{P}}(\Gamma) \\ \swarrow p \uparrow i & \mapsto \swarrow p \uparrow i \end{array} \qquad \begin{array}{rll} \mathsf{lf}^T(\Gamma) : \mathbb{Z} & \to T_{\mathrm{L}}(\Gamma) \\ k & \mapsto \mathsf{lf}(k) \end{array}$$

$$\mathsf{bin}^T(\Gamma) : T_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}), \Gamma) \times T_\tau(\mathrm{B}(\sigma,\mathrm{E}), \Gamma) \to T_{\mathrm{B}(\sigma,\tau)}(\Gamma); \qquad s,\, t \mapsto \mathsf{bin}(s, t).$$

□

The set $T_\tau(\langle\rangle)$ is the set of all complete (i.e. no dangling pointers) cyclic sharing trees of a shape τ.

This development of an initial algebra characterisation follows the line of [FPT99, Fio02, MS03]. Therefore, we can further develop a full theory of algebraic models of abstract syntax for cyclic sharing structures along this line. It will provide second-order typed abstract syntax with object/meta-level variables and substitutions via a substitution monoidal structure and a free Σ-monoid [Ham04, Fio08] in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ (by incorporating suitable arrows into $\mathbb{T}^*$). Object/meta-substitutions on cyclic sharing structures will provide ways to construct cyclic sharing structures from smaller structures in a sensible manner. But this is not the main purpose of this paper. Details will therefore be pursued elsewhere.

3.2. **Structural recursion principle.** An important benefit of initial algebra characterisation is that the unique homomorphism from the initial to another algebra is a mapping defined by structural recursion.

Theorem 3.2. *The unique homomorphism ϕ from the initial Σ-algebra T to a Σ-algebra A is described as*

$$\begin{aligned}
&\phi_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = \mathsf{ptr}^A(\Gamma)(\swarrow p{\uparrow}i)\\
&\phi_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) = \mathsf{lf}^A(\Gamma)(k)\\
&\phi_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) = \mathsf{bin}^A(\Gamma)(\phi_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s),\ \phi_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t)).
\end{aligned}$$

Proof. Since the unique homomorphism $\phi : T \longrightarrow A$ is a morphism of $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. □

Example 3.3. We give examples of functions on T defined by structural recursion.

(i) The function leaves that collects all leaf values in a cyclic sharing tree $t \in T_\tau(\Gamma)$:

$$\begin{aligned}
&\mathsf{leaves} : T \longrightarrow K_{\mathcal{P}(\mathbb{Z})}\\
&\mathsf{leaves}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) &&= \varnothing\\
&\mathsf{leaves}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) &&= \{k\}\\
&\mathsf{leaves}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) &&= \mathsf{leaves}_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s) \cup \mathsf{leaves}_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t).
\end{aligned}$$

This is because leaves is the unique homomorphism from T to a Σ-algebra $K_{\mathcal{P}(\mathbb{Z})}$ (the constant bifunctor to the power set of integers $\mathbb{Z}$) whose operations are given by

$$\begin{aligned}
&\mathsf{ptr}^{K_{\mathcal{P}(\mathbb{Z})}}(\Gamma)(\swarrow p{\uparrow}i) &&= \varnothing\\
&\mathsf{lf}^{K_{\mathcal{P}(\mathbb{Z})}}(\Gamma)(k) &&= \{k\}\\
&\mathsf{bin}^{K_{\mathcal{P}(\mathbb{Z})}}(\Gamma)(x,y) &&= x \cup y.
\end{aligned}$$

(ii) The function height that computes the height of a cyclic sharing tree t:

$$\begin{aligned}
&\mathsf{height} : T \longrightarrow K_{\mathbb{Z}}\\
&\mathsf{height}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) &&= 1\\
&\mathsf{height}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) &&= 1\\
&\mathsf{height}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) &&= \max(\mathsf{height}_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s), \mathsf{height}_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t)) + 1
\end{aligned}$$

where max is the maximum function on $\mathbb{Z}$. This is because height is the unique homomorphism from T to a Σ-algebra $K_{\mathbb{Z}}$ whose algebra structure is the obvious one. Notice that the height is not so directly defined in ordinary graph representations.

(iii) The function $\mathsf{skeleton}$ that computes the shape of a given cyclic sharing tree t:

$$\begin{aligned}
&\mathsf{skeleton} : T \longrightarrow K_{\mathbb{T}}\\
&\mathsf{skeleton}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) &&= \mathrm{P}\\
&\mathsf{skeleton}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) &&= \mathrm{L}\\
&\mathsf{skeleton}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) &&= \mathrm{B}(\mathsf{skeleton}_{\sigma}(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s),\ \mathsf{skeleton}_{\tau}(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t)).
\end{aligned}$$

From an algorithmic perspective, the structural recursion principle provides depth-first search traversal of a rooted graph. Consequently, graph algorithms based on depth-first search are directly programmable using this structural recursion. On the author's home page (`http://www.cs.gunma-u.ac.jp/~hamana/`), several other simple graph algorithms have been programmed using structural recursion.

3.3. **Structural induction principle.** Another important benefit of initial algebra characterisation is the tight connection to structural induction principle. To derive it, following [HJ98, Jac99], we use the category $\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$ of predicates on $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ defined by

- objects: sub-presheaves $(P \hookrightarrow U)$, i.e., inclusions between $P, U \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$,
- arrows: $u : (Q \hookrightarrow V) \to (P \hookrightarrow U)$ are natural transformations $u : V \to U$ between underlying presheaves satisfying $a \in Q_\tau(\Gamma)$ implies $u(a) \in P_\tau(\Gamma)$ for all $\tau \in \mathbb{T}, \Gamma \in \mathbb{T}^*$.

A sub-presheaf $(P \hookrightarrow T)$ is seen as a predicate P on cyclic sharing terms T, which is indexed by types and contexts. So, we say "$P^{\Gamma}_{\tau}(t)$ holds" when $t \in P_\tau(\Gamma)$ for $t \in T_\tau(\Gamma)$.

We consider Σ-algebras in $\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$ by "logical predicate" lifting [HJ98] of algebras in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Why this is lifting is that now we consider the functor $p : \mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}) \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ sending $(P \hookrightarrow U)$ to the underlying presheaf U. Then we lift the functor Σ to Σ_{pred} in a commuting diagram

$$\begin{tikzcd}
\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}) \arrow[r, "\Sigma_{\text{pred}}"] \arrow[d, "p"'] & \mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}) \arrow[d, "p"] \\
(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \arrow[r, "\Sigma"'] & (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}
\end{tikzcd}$$

by induction on the structure of Σ:

$$\begin{aligned}
(\Sigma_{\text{pred}}(P \hookrightarrow U))_{\mathrm{P}} &= (\mathrm{PO} \hookrightarrow \mathrm{PO})\\
(\Sigma_{\text{pred}}(P \hookrightarrow U))_{\mathrm{E}} &= (0 \hookrightarrow 0)\\
(\Sigma_{\text{pred}}(P \hookrightarrow U))_{\mathrm{L}} &= (K_{\mathbb{Z}} \hookrightarrow K_{\mathbb{Z}})\\
(\Sigma_{\text{pred}}(P \hookrightarrow U))_{\mathrm{B}(\sigma,\tau)} &= \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})}(P \hookrightarrow U)_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})}(P \hookrightarrow U)_\tau
\end{aligned}$$

where we also lift the context extension to $\delta_\tau : \mathrm{Sub}(\mathbf{Set}^{\mathbb{T}^*}) \to \mathrm{Sub}(\mathbf{Set}^{\mathbb{T}^*})$ defined by $\delta_\tau(A \hookrightarrow B) = (A\langle \tau, -\rangle \hookrightarrow B\langle \tau, -\rangle)$.

A Σ_{pred}-algebra structure $\alpha : \Sigma_{\text{pred}}(P \hookrightarrow T) \to (P \hookrightarrow T)$ can be read as the induction steps in a proof by structural induction. For example, the operation in $\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$

$$\begin{aligned}\mathsf{bin}^{\sigma,\tau\, P}(\Gamma) : (P_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma) \hookrightarrow T_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)) \times (P_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma) \hookrightarrow T_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma))\\ \to (P_{\mathrm{B}(\sigma,\tau)}(\Gamma) \hookrightarrow T_{\mathrm{B}(\sigma,\tau)}(\Gamma))\\ s,t \mapsto \mathsf{bin}(s,t)\end{aligned}$$

means that "if $P_\sigma^{\Gamma,\mathrm{B}(\mathrm{E},\mathrm{E})}(s)$ & $P_\tau^{\Gamma,\mathrm{B}(\sigma,\mathrm{E})}(t)$ holds, then $P^{\Gamma}_{\mathrm{B}(\sigma,\tau)}(\mathsf{bin}(s,t))$ holds."

Jacobs showed that if a fibration $\mathcal{E} \to \mathcal{B}$ satisfies several conditions (having fibered (co)products, etc.), then the logical predicate lifting from $\mathcal{B}$ to $\mathcal{E}$ preserves initial algebras (Prop. 9.2.7 in [Jac99]). The functor $p : \mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}) \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ is actually such a fibration. Consequently, because T is an initial Σ-algebra, $(T \hookrightarrow T)$ is an initial Σ_{pred}-algebra. The unique homomorphism $\phi : (T \hookrightarrow T) \longrightarrow (P \hookrightarrow T)$ means that P holds for all cyclic sharing terms in T. Hence

Theorem 3.4. *Let P be a predicate on T. To prove that $P^\Gamma_\tau(t)$ holds for all $t \in T_\tau(\Gamma)$, it suffices to show*
(i) *$P^\Gamma_{\mathrm{P}}(\swarrow p{\uparrow}i)$ holds for all $\swarrow p{\uparrow}i \in \mathrm{PO}(\Gamma)$,*
(ii) *$P^\Gamma_{\mathrm{L}}(\mathsf{lf}(k))$ holds for all $k \in \mathbb{Z}$,*
(iii) *if $P_\sigma^{\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma}(s)$ & $P_\tau^{\mathrm{B}(\sigma,\mathrm{E}),\Gamma}(t)$ holds, then $P^{\Gamma}_{\mathrm{B}(\sigma,\tau)}(\mathsf{bin}(s,t))$ holds.*

This structural induction principle is useful to prove properties of functions on cyclic sharing terms defined by structural recursion. As an example, we show the following simple property of the function skeleton defined in Example 3.3.

Proposition 3.5. *For all $t \in T_\tau(\Gamma)$, $\mathsf{skelton}_\tau(\Gamma)(t) = \tau$.*

Proof. By structural induction on t.
(i) Case $t = \swarrow p{\uparrow}i \in \mathrm{PO}(\Gamma)$. By definition, $\mathsf{skelton}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = \mathrm{P}$.
(ii) Case $t = \mathsf{lf}(k)$. By definition, $\mathsf{skelton}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) = \mathrm{L}$.
(iii) Case $t = \mathsf{bin}(s_1, s_2)$. Then,

$$\begin{aligned}\mathsf{skeleton}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s_1,s_2)) &= \mathrm{B}(\mathsf{skeleton}_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s_1), \mathsf{skeleton}_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(s_2))\\ &= \mathrm{B}(\sigma,\tau) \qquad \text{by induction hypothesis.}\end{aligned}$$ □

4. Inductive Types for Cyclic Sharing Structures

In this section, we achieve our goal [II] to give inductive types for cyclic sharing structures. We give implementations in two different systems. We first use the functional language Haskell for an implementation because
(i) we show that our characterisation of cyclic sharing is available in today's programming language technology, and
(ii) Haskell's type system is powerful enough to implement our initial algebra characterisation faithfully.

Secondly, we give an implementation by dependent types in the proof assistant Agda.

4.1. **A GADT definition in Haskell.** Because the set $T_\tau(\Gamma)$ of cyclic sharing terms depends on a shape tree and context, it should be implemented as a dependent type. We have seen in the proof of Theorem 3.1 that constructors of cyclic sharing terms are $\mathbb{T}$ and $\mathbb{T}^*$-indexed functions. Inductive types defined by indexed constructors have been known as *inductive families* in dependent type theories [Dyb94]. Recently, the Glasgow Haskell Compiler (GHC) incorporates this feature as *GADTs* (generalised algebraic data types) [PVWW06]. Using another feature called type classes, we can realise lightweight dependently-typed programming in Haskell [McB02].

We will implement $T_\tau(\Gamma)$ as a GADT "`T n t`" that depends on a context `n` (for Γ) and a shape tree `t` (for τ). In Haskell, a type can only depend on types (not values). For that reason, we firstly define type-level shape trees by using a type class.

```
  data E
  data P
  data L     = StopLf
  data B a b = DnL a | DnR b | StopB

  class Shape t
  instance Shape E
  instance Shape P
  instance Shape L
  instance (Shape s, Shape t) => Shape (B s t)
```

These define constructors of shape trees as types `E`,`P`,`L` and a type constructor `B`, then group them by the type class `Shape`. Values of a shape tree type τ are defined by $\mathcal{P}os(\tau)$, i.e. "referable positions" in τ. For example, consider a shape tree $\mathrm{B}(\mathrm{B}(\mathrm{L},\mathrm{L}),\mathrm{L})$. The position $1\cdot 2$ in this shape tree is coded as the well-typed term `DnL (DnR StopLF) :: B (B L L) L` where `StopLf` means stopping at a leaf.

Similarly, a typing context $\langle\tau_1,\ldots,\tau_n\rangle$ is coded as a type-level sequence

$$\texttt{TyCtx}\ \tau_1\ (\texttt{TyCtx}\ \tau_2\cdots(\texttt{TyCtx}\ \tau_n\ \texttt{TyEmp}))$$

and the type constructors are grouped by the type class `Ctx`. Values of a context type are "pointers" (e.g. `(Up UpStop)` meaning $\uparrow$2).

```
  data TyEmp
  data TyCtx t n = Up n | UpStop | UpGD t

  class Ctx n
  instance Ctx TyEmp
  instance (Shape t, Ctx n) => Ctx (TyCtx t n)
```

Finally, we define the set $T_\tau(\Gamma)$ as a GADT "`T`" that takes a context and a shape tree as two arguments of the type constructor `T`.

```
data T :: * -> * -> * where
  Ptr :: Ctx n => n -> T n P
  Lf  :: Ctx n => Int -> T n L
  Bin :: (Ctx n, Shape s, Shape t) =>
         T (TyCtx (B E E) n) s -> T (TyCtx (B s E) n) t -> T n (B s t)
```

This defines three constructors of cyclic sharing terms faithfully. Note that the part "`Ctx n =>`", called a context of a type class, is a quantification meaning that "for every type `n` which is an instance of the type class `Ctx`".

For example, the term in Example 2.1 is certainly a well-typed term; its type is inferred in the GHC (by invoking the command `:t` in the interpreter)

```
Bin (Bin (Lf 5) (Lf 6))
    (Bin (Ptr (Up (UpGD (DnL (DnL StopLf))))) (Lf 7))
:: T TyEmp (B (B L L) (B P L))
```

The term `Up (UpGD (DnL (DnL StopLf)))` is the representation of the pointer $\swarrow 11 \uparrow 2$ in de Bruijn notation, which is read from the top as "going up and up, then going down (`GD` is short for going down) to the position 11 and stopping at a leaf". The type inference and the type checker automatically ensure well-formedness of cyclic sharing terms.

In Haskell, we can equally use the GADT `T` as an ordinary algebraic datatype. Therefore, we can define a function on it by structural recursion as described in Example 3.3 (even simpler; shape tree and context parameters are unnecessary in defining functions because of Haskell's compilation method [PVWW06]). The implementation and additional examples using the GADT `T` are available from the author's home page.

4.2. A dependent type definition in Agda.

Secondly, we consider a definition in a proof assistant/dependently-typed programming language Agda [Nor07]. There are several ways for implementation. One way is to use so-called universe construction [OS08] by defining decoding functions from type names to actual types to mimic the type class mechanism used in the previous subsection. The resulting definition might resemble the Haskell version. Another way is more natural to use the full power of dependent types in Agda. In this subsection, we take this approach. We implement the initial algebra T of cyclic sharing tree structures as a dependent type that depends on two *values* (not types as in Haskell), a shape tree and a context.

We maximally use Agda's notational advantage, which allows Unicode for mathematical symbols in a program. In the following Agda code, we use mathematical symbols we have used in the paper to the greatest degree possible (but it is certainly a real Agda code, not a pseudo-code).

We define shape trees as a usual inductive type, and contexts as the type of sequences of shape trees (where ■ is the empty context and "," is the separator):

```
data Shape : Set where
    E : Shape
    P : Shape
    L : Shape
    B : Shape → Shape → Shape

data Ctx : Set where
    ■   : Ctx
    _,_ : Shape → Ctx → Ctx
```

The type $Pos\, \tau$ for positions of a shape tree τ is defined naturally as an Agda's inductive family, which consists of indexed constructors. The style of definition is almost identical to that of GADTs in Haskell.

$$\begin{array}{l} \textbf{data}\ \mathit{Pos}\ :\ \mathit{Shape}\ \to\ \mathit{Set}\ \textbf{where} \\ \quad \epsilon \quad\ :\ \mathit{Pos}\ L \\ \quad \epsilon' \quad :\ \forall\,\{\sigma\ \tau\}\ \to\ \mathit{Pos}\,(B\ \sigma\ \tau) \\ \quad \mathit{DnL}\ :\ \forall\,\{\sigma\ \tau\}\ \to\ \mathit{Pos}\ \sigma\ \to\ \mathit{Pos}\,(B\ \sigma\ \tau) \\ \quad \mathit{DnR}\ :\ \forall\,\{\sigma\ \tau\}\ \to\ \mathit{Pos}\ \tau\ \to\ \mathit{Pos}\,(B\ \sigma\ \tau) \end{array}$$

To define the dependent type T for cyclic sharing trees, the crucial ingredient is the implementation of the presheaf PO for pointers, recalling that it was defined by

$$\mathrm{PO}(\langle\tau_1,\ldots,\tau_n\rangle) = \{\swarrow p{\uparrow}i \mid 1 \le i \le n,\ p \in \mathcal{P}os(\tau_i)\}.$$

What we need is to implement a way to pick an index i and τ_i from a typing context concisely. The following type *Index* does this job.

$$\begin{array}{l} \textbf{data}\ \mathit{Index}\ :\ \mathit{Shape}\ \to\ \mathit{Ctx}\ \to\ \mathit{Set}\ \textbf{where} \\ \quad \mathit{one}\ :\ \forall\,\{\Gamma\ \tau\}\ \to\ \mathit{Index}\ \tau\ (\tau\ ,\ \Gamma) \\ \quad s \quad\ :\ \forall\,\{\Gamma\ \tau\ \sigma\}\ \to\ \mathit{Index}\ \tau\ \Gamma\ \to\ \mathit{Index}\ \tau\ (\sigma\ ,\Gamma) \end{array}$$

A well-typed term "$i\ :\ \mathit{Index}\ \tau\ \Gamma$" means "$i$ is the index of a shape tree τ in $\Gamma = \tau_1,\ldots,\tau,\ldots,\tau_n$", e.g., $s\ (s\ \mathit{one})\ :\ \mathit{Index}\ \tau_3\ (\tau_1\ ,\ \tau_2\ ,\ \tau_3\ ,\ \blacksquare)$. Then, the presheaf PO is naturally implemented.

$$\begin{array}{l} \textbf{data}\ \mathit{PO}\ :\ \mathit{Ctx}\ \to\ \mathit{Set}\ \textbf{where} \\ \quad \swarrow\_{\uparrow}\_\ :\ \forall\,\{\Gamma\ \tau\}\ \to\ \mathit{Pos}\ \tau\ \to\ \mathit{Index}\ \tau\ \Gamma\ \to\ \mathit{PO}(\Gamma) \end{array}$$

Using these ingredients, the implementation of the initial algebra T for cyclic sharing trees is quite the same as the mathematical definition we obtained in Theorem 3.1.

$$\begin{array}{l} \textbf{data}\ T\ :\ \mathit{Ctx}\ \to\ \mathit{Shape}\ \to\ \mathit{Set}\ \textbf{where} \\ \quad \mathit{ptr}\ :\ \forall\,\{\Gamma\}\ \to\ \mathit{PO}(\Gamma)\ \to\ T\ \Gamma\ P \\ \quad \mathit{lf} \quad :\ \forall\,\{\Gamma\}\ \to\ \mathit{Int}\ \to\ T\ \Gamma\ L \\ \quad \mathit{bin}\ :\ \forall\,\{\Gamma\ \sigma\ \tau\}\ \to\ T\ (B\ E\ E\ ,\ \Gamma)\ \sigma\ \to\ T\ (B\ \sigma\ E\ ,\ \Gamma)\ \tau\ \to\ T\ \Gamma\ (B\ \sigma\ \tau) \end{array}$$

For example, the term in Example 2.1 is a well-typed term also in Agda.

$$\mathit{bin}\ (\mathit{bin}\ (\mathit{lf}\ 5)\ (\mathit{lf}\ 6))\ (\mathit{bin}\ (\mathit{ptr}\ (\swarrow\ \mathit{DnL}\ (\mathit{DnL}\ \epsilon)\uparrow s\ \mathit{one})\ (\mathit{lf}\ 7)))\ :\ T\,\blacksquare\,(B\ (B\ L\ L)\ (B\ P\ L))$$

Defining a function on the type T by structural recursion is directly possible because of Agda's pattern matching mechanism on dependent types. The functions in Example 3.3 are defined directly. In addition, shape tree and context parameters can be (Agda's feature of) implicit arguments. Therefore, we can use such functions concisely by omitting complex indices, as in the case of GADTs in Haskell.

5. General Signature

We give construction of cyclic sharing structures for arbitrary signatures as a natural generalisation of the binary tree case.

A *signature* Σ for cyclic sharing structures consists of a set Σ of function symbols having arities. A function symbol of arity $n \in \mathbb{N}$ is denoted by $f^{(n)}$. Each function symbol f has an associated *shape symbol* $\lceil f \rceil$ (typically written in small caps such as B).

Example 5.1. For the case of cyclic sharing binary trees, the signature Σ consists of $\mathsf{bin}^{(2)}$ and $\mathsf{lf}^{(0)}$. Corresponding shape symbols are defined by $\lceil\mathsf{bin}\rceil = \textsc{b}$, $\lceil\mathsf{lf}\rceil = \textsc{l}$.

The set $\mathbb{T}$ of all shape trees is defined by

$$\mathbb{T} \ni \tau ::= \mathrm{E} \mid \mathrm{P} \mid \lceil f \rceil(\tau_1, \ldots, \tau_n) \quad \text{for each } f^{(n)} \in \Sigma.$$

The set of all contexts is

$$\mathbb{T}^* = \{\langle \tau_1, \ldots, \tau_n \rangle \mid n \in \mathbb{N},\, i \in \{1, \ldots, n\},\, \tau_i \in \mathbb{T}\}.$$

Positions are defined by

$$\mathcal{P}os(\mathrm{E}) = \mathcal{P}os(\mathrm{P}) = \varnothing$$
$$\mathcal{P}os(\lceil f \rceil(\tau_1, \ldots, \tau_n)) = \{\epsilon\} \cup \{1.p \mid p \in \mathcal{P}os(\tau_1)\} \cup \ldots \cup \{n.p \mid p \in \mathcal{P}os(\tau_n)\}.$$

Typing rules

$$\frac{|\Gamma| = i - 1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma, \sigma, \Gamma' \vdash \swarrow p \!\uparrow\! i : \mathrm{P}} \qquad \frac{\gamma_1, \Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \gamma_n, \Gamma \vdash t_n : \tau_n \quad f^{(n)} \in \Sigma}{\Gamma \vdash f(t_1, \ldots, t_n) : \lceil f \rceil(\tau_1, \ldots, \tau_n)}$$

where $\gamma_1 = \lceil f \rceil(\mathrm{E}, \ldots, \mathrm{E})$, $\gamma_{i+1} = \lceil f \rceil(\tau_1, \ldots, \tau_i, \mathrm{E}, \ldots, \mathrm{E})$ for each $1 \le i \le n-1$.

The shape trees γ_i's are also used below.

This general case has the safety and uniqueness properties as well.

Theorem 5.2 (Safety). *If a closed term $\vdash t : \tau$ is derivable, any pointer in t points to a node in t.*

Theorem 5.3 (Uniqueness). *Given a rooted graph that is connected, directed and edge-ordered, the term representation is unique.*

Next we provide an initial algebra characterisation. The base category is $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. The presheaf PO of pointers is defined the same as in Sect. 3. For a signature Σ, we associate a signature functor $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ defined by

$$(\Sigma A)_{\mathrm{E}} = 0 \qquad (\Sigma A)_{\mathrm{P}} = \mathrm{PO} \qquad (\Sigma A)_{\lceil f \rceil(\tau_1, \ldots, \tau_n)} = \prod_{1 \le i \le n} \delta_{\gamma_i} A_{\tau_i} \quad \text{for each } f^{(n)} \in \Sigma.$$

The following theorems are straightforward generalisations of the corresponding theorems for the binary tree case; hence proofs are straightforward.

Theorem 5.4 (Initial algebra). *Let Σ be a signature. $T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}$ forms an initial Σ-algebra where operations are:*

$$\begin{array}{llll} \mathsf{ptr}^T(\Gamma) : \mathrm{PO}(\Gamma) & \to T_{\mathrm{P}}(\Gamma) & \qquad f^T(\Gamma) : \prod_{1 \le i \le n} T_{\tau_i}(\gamma_i, \Gamma) & \to T_{\lceil f \rceil(\tau_1, \ldots, \tau_n)}(\Gamma) \\ \swarrow p \!\uparrow\! i & \mapsto \swarrow p \!\uparrow\! i & \qquad t_1, \ldots, t_n & \mapsto f(t_1, \ldots, t_n). \end{array}$$

Theorem 5.5 (Structural recursion). *The unique homomorphism ϕ from the initial Σ-algebra T to a Σ-algebra A is described as*

$$\phi_{\mathrm{P}}(\Gamma)(\swarrow p \!\uparrow\! i) = \mathsf{ptr}^A(\Gamma)(\swarrow p \!\uparrow\! i)$$
$$\phi_{\lceil f \rceil(\tau_1, \ldots, \tau_n)}(\Gamma)(f(t_1, \ldots, t_n)) = f^A(\Gamma)(\phi_{\tau_1}(\gamma_1, \Gamma)(t_1), \ldots, \phi_{\tau_n}(\gamma_n, \Gamma)(t_n)).$$

Theorem 5.6 (Structural induction). *To prove that $P^\Gamma_\tau(t)$ holds for all $t \in T_\tau(\Gamma)$, it suffices to show*

(i) *$P^\Gamma_{\mathrm{P}}(\swarrow p \!\uparrow\! i)$ holds for all $\swarrow p \!\uparrow\! i \in \mathrm{PO}(\Gamma)$,*

(ii) *if* $f^{(n)} \in \Sigma$ *and* $P^{\gamma_i,\Gamma}_{\tau_i}(t_i)$ *holds for all* $i = 1,\dots,n$, *then* $P^{\Gamma}_{\lceil f \rceil(\tau_1,\dots,\tau_n)}(f(t_1,\dots,t_n))$ *holds.*

Moreover, to give a GADT in Haskell and a dependent type in Agda for cyclic sharing structures of a given signature is straightforward, along the line of Sect. 4 for the signature of binary cyclic sharing trees.

6. Variations of the Form of Pointers

We have concentrated up to this point on *unique representation* of a given rooted graph. This has been achieved by imposing the form of pointers in cyclic sharing trees only *from right to left*. In this section, we consider relaxation of this restriction as a variation of the theme of the paper.

Actually, our algebraic framework, algebras in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$, is not only for depth-first search trees. Algebras in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ can model *trees with arbitrary pointers*, and the form of pointers can be controlled by types using shape trees. That is, our framework has sufficient flexibility to represent any form of pointers precisely. In addition, from the application perspective, other forms of pointers will be useful. For example, one may need to invert pointers in a cyclic sharing tree in some algorithms. In such a case, one needs to use pointers *from left to right*, not only from right to left.

Syntactically, the form of pointers is determined by shape tree types of function symbols and the definition of position function $\mathcal{P}os$. Consequently, relaxing the restrictions is an easy modifications of the previous treatment. Semantically, such variations of signature give other algebras of functors in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ for trees with pointers of various forms.

6.1. **Left-to-right pointers.** First, we consider cyclic sharing trees involving left-to-right pointers and not involving right-to-left pointers. An example is the tree (i) in Fig. 3, as represented by $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5), \swarrow 21{\uparrow}2), \mathsf{bin}(\mathsf{lf}(8), \mathsf{lf}(7)))$. This case retains the uniqueness property of the representation for a given rooted graph.

A signature Σ, types $\mathbb{T}$, contexts $\mathbb{T}^*$ and positions $\mathcal{P}os$ are defined exactly the same as they are in Sect. 5.

Typing rules

$$\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma, \sigma, \Gamma' \vdash \swarrow p{\uparrow}i : \mathrm{P}} \qquad \frac{\gamma_1, \Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \gamma_n, \Gamma \vdash t_n : \tau_n \quad f^{(n)} \in \Sigma}{\Gamma \vdash f(t_1,\dots,t_n) : \lceil f \rceil(\tau_1,\dots,\tau_n)}$$

where $\gamma_n = \lceil f \rceil(\mathrm{E},\dots,\mathrm{E})$, $\gamma_i = \lceil f \rceil(\mathrm{E},\dots,\mathrm{E},\tau_{i+1},\dots,\tau_n)$ for each $1 \le i \le n-1$.

The category $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$, a signature functor $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$, and the Σ-initial algebra T are also defined similarly to the definitions in Sect. 5. Associated structural recursion and induction follow as well.

Example 6.1. Consider the case of cyclic sharing binary trees involving left-to-right pointers. The difference from the original typing rules is the case of a binary node. Now, the above rule is instantiated as

$$\frac{\mathrm{B}(\mathrm{E},\tau), \Gamma \vdash s : \sigma \quad \mathrm{B}(\mathrm{E},\mathrm{E}), \Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{bin}(s,t) : \mathrm{B}(\sigma,\tau)}$$

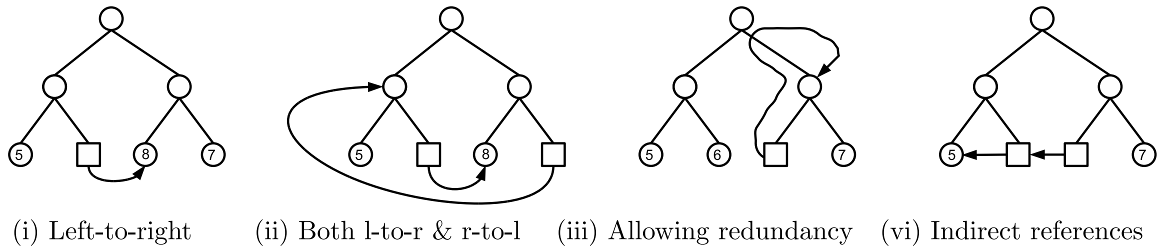


Figure 3: Trees involving various pointers

This means that the shape tree type $\mathrm{B}(\mathrm{E},\tau)$ at the left on the upper judgments expresses that (a pointer in) s can point to a node in t, whereas the shape tree type $\mathrm{B}(\mathrm{E},\mathrm{E})$ at the right expresses that (a pointer in) t cannot point to any node in s by masking node information of s by the void shape E. Actually, the general typing rule for left-to-right pointers is obtained by generalising this observation.

6.2. **Symmetric form of pointers.** We can further allow both right-to-left and left-to-right pointers. An example is the tree (ii) in Fig. 3 represented by $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5),\swarrow 21\uparrow 2),\mathsf{bin}(\mathsf{lf}(8),\swarrow 1\uparrow 2))$. The only difference is the typing rules.

Typing rules

$$\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma,\sigma,\Gamma' \vdash \swarrow p\uparrow i : \mathrm{P}} \qquad \frac{\gamma_1,\Gamma \vdash t_1:\tau_1 \quad \cdots \quad \gamma_n,\Gamma \vdash t_n:\tau_n \quad f^{(n)}\in\Sigma}{\Gamma \vdash f(t_1,\ldots,t_n) : \lceil f\rceil(\tau_1,\ldots,\tau_n)}$$

where $\gamma_i = \lceil f\rceil(\tau_1,\ldots,\tau_{i-1},\mathrm{E},\tau_{i+1},\ldots,\tau_n)$ for each $1 \le i \le n$ (i.e. only i-th argument is set as E).

A shape tree $\gamma_i = \lceil f\rceil(\tau_1,\ldots,\tau_{i-1},\mathrm{E},\tau_{i+1},\ldots,\tau_n)$ is used to prohibit only redundant reference (i.e. going up to an upper node and then going back down through the same path) by the void shape E.

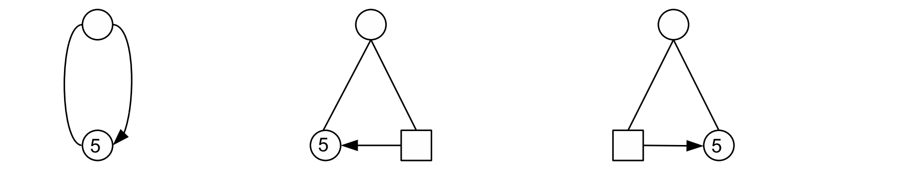
This case has no uniqueness property for a given graph, because for example, a graph in Fig. 4 (i) can be represented in two ways (ii) and (iii) using cyclic sharing terms.

6.3. **No restriction of pointers.** In addition to the symmetric form of pointers, redundant references can be allowed. An example is the tree (iii) in Fig. 3 represented by $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5),\mathsf{lf}(6)),\mathsf{bin}(\swarrow 2\uparrow 2,\mathsf{lf}(7)))$. Redundant reference means that the path obtained by $\swarrow p\uparrow i$ is not the shortest path to the destination. In the case of the tree (iii) in Fig. 3, going up to the root and then going down through the same path to the right child.

Typing rules

$$\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma,\sigma,\Gamma' \vdash \swarrow p\uparrow i : \mathrm{P}} \qquad \frac{\gamma,\Gamma \vdash t_1:\tau_1 \quad \cdots \quad \gamma,\Gamma \vdash t_n:\tau_n \quad f^{(n)}\in\Sigma}{\Gamma \vdash f(t_1,\ldots,t_n) : \lceil f\rceil(\tau_1,\ldots,\tau_n)}$$

where $\gamma = \lceil f\rceil(\tau_1,\ldots,\tau_n)$.



(i) A graph (ii) Using right-to-left pointer (iii) Using left-to-right pointer

Figure 4: Two representations of a graph

6.4. **Allowing indirect references.** Up to this point in the discussion, we have assumed that a pointer cannot point to another pointer node. Like the tree (vi) in Fig. 3 has been prohibited because we aimed to obtain a unique representation for a graph. However, that assumption can also be relaxed. This is achieved by merely modifying the definition of $\mathcal{P}os$ as

$$\mathcal{P}os(\mathrm{P}) = \{\epsilon\}.$$

The tree (vi) in Fig. 3 is represented by $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5), \swarrow 1\uparrow 1), \mathsf{bin}(\swarrow 12\uparrow 2, \mathsf{lf}(7)))$.

6.5. **Pointers from inner nodes.** We can allow pointers from inner nodes, i.e., not only from leaves as we have considered. This is by introducing a new term construct

$$f(\swarrow p\uparrow i; t_1, \ldots, t_n)$$

which expresses that an inner node f having n-children also has a pointer slot. Typing rule is the combination of the previous term formations for the pointer and function term.

Typing rule

$$\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma) \quad \gamma, \Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \gamma, \Gamma \vdash t_n : \tau_n \quad f^{(n)} \in \Sigma}{\Gamma, \sigma, \Gamma' \vdash f(\swarrow p\uparrow i; t_1, \ldots, t_n) : \lceil f \rceil(\tau_1, \ldots, \tau_n)}$$

where $\gamma_i = \lceil f \rceil(\tau_1, \ldots, \tau_{i-1}, \mathrm{E}, \tau_{i+1}, \ldots, \tau_n)$ for each $1 \le i \le n$.

This form of terms is used as a data model of XML called *trees with pointers* [CGZ05] by Calcagno, Gardner and Zarfaty.

6.6. **Mixing variations.** The form of pointers need not be uniform (i.e. all pointers must be the same form) as described above. For example, in a single tree, it is possible that some function symbols allow only right-to-left pointers, some others allow only left-to-right, and some others allow both, etc. This possibility is realised merely by assigning an appropriate type to each function symbol, which shows that our framework has expressive power to control the form of pointers.

It is important to note that in any variation of typing rules, the safety property of pointers stated in Theorem 5.2 still holds. Truly dangling pointers cannot happen in this framework.

7. Connection to Equational Term Graphs in the Initial Algebra Framework

We have investigated a term syntax for cyclic sharing structures, which gives a representation of a graph. In this section, we give the converse, i.e., an explicit way to calculate the graph for a given cyclic sharing term. This means to give semantics of a cyclic sharing term by a finite graph. We give it using Ariola and Klop's equational term graphs in the initial algebra framework. This semantics also clarifies connections to existing works that have explored the semantics of cyclic sharing structures.

Equational term graphs [AK96] are another representation of cyclic sharing trees, which have been used in a formulation of term graph rewriting. This is a representation of a rooted graph[1] by associating a unique name to each node and by writing down the interconnections through a set of recursive equations. For example, the graph portrayed in Figure 5 is represented as an equational term graph

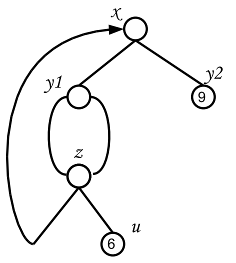


Figure 5: rooted graph

$$\{x \mid \begin{array}{llll} x & = \mathsf{bin}(y_1, y_2), & y_2 & = \mathsf{lf}(9), \\ y_1 & = \mathsf{bin}(z, z), & z & = \mathsf{bin}(x, u), \\ u & = \mathsf{lf}(6)\}. \end{array}$$

We use this form of equational term graphs, which is called flattened form in [AK96], and which is formally defined as follows (NB. it differs slightly from the original syntax to make explicit the connection to cyclic sharing terms).

Suppose a signature Σ and a set $X = \{x, x_1, \ldots\}$ of variables. *An equational term graph* is of the form

$$\{x \mid x_1 = t_1,\, x_2 = t_2, \ldots\}$$

where each t_i follows the syntax

$$t ::= x \mid \swarrow p{\uparrow}i \mid f(x_1, \ldots, x_n).$$

A variable is called *bound* if it appears in the left-hand side of an equation; it is called *free* otherwise. We also call $\swarrow p \uparrow i$ a *free variable* (and regard it as a free variable). We assume that any useless equation $y = t$, where y cannot be reachable from the root, is automatically removed in the presentation of equational term graphs [AK96] (hence, equational term graphs are always connected and single-rooted).

We define a translation from a cyclic sharing term to an equational term graph by the unique homomorphism from the initial algebra to an algebra consisting of equational term graphs. The idea is to use positions as unique variables in an equational term graph. We define $\mathsf{EGraph}_\tau(\Gamma)$ by the set of all equational term graphs having free variables taken from $\mathrm{PO}(\Gamma)$ (where a shape index τ is meaningless for equational term graphs, but we just put this index to form a presheaf). This EGraph forms a presheaf in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Any equational term graph can be drawn as a tree-like graph as Fig. 5 by traversing each node in a depth-first search manner from the root. Therefore, we can assign each node to its position in the whole equational term graph. Consequently, an equational term graph

$$\{x_1 \mid x_1 = t_1,\, x_2 = t_2, \ldots\}$$

[1]For the case of an unrooted graph, it can be rooted by choosing an arbitrary starting node. It may also have several other distinct connected components, which might be represented by a set of equational term graphs.

can be normalised to an "α-normal form" in which for each $x = t$, the bound variable x is renamed to the position of t in the whole term as

$$\{\epsilon \mid \epsilon = t'_1,\ 1 = t'_2, \ldots\}$$

(see Example 7.2). We identify an equation term graph with its α-normal form.

Proposition 7.1. EGraph *forms a* Σ*-algebra. The unique homomorphism* $[\![-]\!] : T \longrightarrow \mathsf{EGraph}$ *is monomorphic, giving an interpretation of a cyclic sharing term as a graph represented by an equational term graph.*

Proof. We define an algebra structure on EGraph of α-normal forms as follows.

$$f_\tau^{\mathsf{EGraph}}(\Gamma)(\{\epsilon \mid G_1\}, \ldots, \{\epsilon \mid G_n\}) = \{\epsilon \mid \epsilon = f(1, \ldots, n),\ G'_1, \ldots, G'_n\}$$
$$\textbf{where } \{1 \mid G'_1\} = \mathsf{shift}_1(\{\epsilon \mid G_1\}) \ \cdots\ \{n \mid G'_n\} = \mathsf{shift}_n(\{\epsilon \mid G_n\})$$
$$\mathsf{ptr}_\tau^{\mathsf{EGraph}}(\Gamma)(\swarrow p\!\uparrow\! i) = \{\epsilon \mid \epsilon = \swarrow p\!\uparrow\! i\}$$

$$\mathsf{shift}_i\{\epsilon \mid \epsilon = t_1,\ 1 = t_2, \ldots\} = \{i \mid i = \mathsf{shift}_i(t_1),\ i.1 = \mathsf{shift}_i(t_2), \ldots\}$$
$$\mathsf{shift}_i(p) = i.p \quad \text{for a position } p$$
$$\mathsf{shift}_i(f(x_1, \ldots, x_n)) = f(i.x_1, \ldots, i.x_n)$$
$$\mathsf{shift}_i(\swarrow p\!\uparrow\! x) = \begin{cases} \swarrow p\!\uparrow\! x-1 & \textbf{if } x > 1 \\ p & \textbf{if } x = 1 \end{cases}$$

The function shift_i shifts every bound variable by a position $i \in \mathbb{N}$ (i.e. appending i as prefix) in a term to form an equational term graph suitably. Then, it is obvious that $[\![-]\!]$ is monomorphic and that it gives a translation from cyclic sharing terms to equational term graphs. □

Notice that $[\![-]\!] : T \longrightarrow \mathsf{EGraph}$ is not an isomorphism. Equational term graphs have much more freedom to express graphs than cyclic sharing terms. For example, although $\{x \mid x = x\}$ is a valid equational term graph (the "black hole"), no corresponding cyclic sharing term exists.

Example 7.2. Consider the term $\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mu z.\mathsf{bin}(\uparrow\! x, \mathsf{lf}(6)), \swarrow 1\!\uparrow\! y_1), \mathsf{lf}(9))$ of Fig. 1. This is represented as the following term in de Bruijn and is interpreted as an equational term graph:

$$\mathsf{bin}(\mathsf{bin}(\mathsf{bin}(\uparrow\! 3, \mathsf{lf}(6)), \swarrow 1\!\uparrow\! 1),\ \mathsf{lf}(9))$$
$$\overset{[\![-]\!]}{\mapsto} \quad \begin{array}{llllllll} \{\epsilon \mid & \epsilon & = \mathsf{bin}(1,2), & 12 & = 11, & 112 & = \mathsf{lf}(6), \\ & 1 & = \mathsf{bin}(11,12), & 111 & = \epsilon, & 2 & = \mathsf{lf}(9), \\ & 11 & = \mathsf{bin}(111,112)\}. \end{array}$$

8. Further Connections to Other Works

The semantics of cyclic sharing terms by equational term graphs opens connections to other semantics as $T \longrightarrow \mathsf{EGraph} \longrightarrow \mathcal{S}$, where $\mathcal{S}$ is any of the following semantics of equational term graphs.

(i) **letrec**-expressions: an equational term graph is obviously seen as a **letrec**-expression[2].

[2] **letrec**-expressions are more expressive than equational term graphs because they can express multiple roots by putting a tuple $(x_1, \ldots, x_n)$ of roots of distinct connected components in the body of a **letrec**-expression [Has97].

(ii) Domain-theoretic semantics: mentioned below.
(iii) Categorical semantics in terms of traced symmetric monoidal categories [Has97].
(iv) Coalgebraic semantics: a graph is regarded as a coalgebraic structure that produces every node information along its edges, e.g. [AAMV03].

The domain-theoretic semantics of **letrec**-expressions or systems of recursive equations (e.g. [CKV74]), is now standard; it gives infinite expansion of cyclic sharing structures. Via equational term graphs, we can interpret our cyclic sharing terms in each of these semantics. Each semantics has its own advantage and principles related to some aspects of cyclic sharing structures. However, *none of these has focused on our goals*, which are the following. [I] A simple term syntax that admits structural induction. [II] Direct usability in functional programming, as described in the Introduction. Therefore, we have chosen the initial algebra approach to cyclic sharing structures.

Although insufficient, the above semantics (i) and (ii) are close to our goals in the following way. Consider the cyclic sharing term $\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mu z.\mathsf{bin}(\uparrow x, \mathsf{lf}(6)), \swarrow 1 \uparrow y_1), \mathsf{lf}(9))$ of Fig. 1. As considered in Example 7.2, this is interpreted as an equational term graph:

$$\begin{array}{lllllllll}\{\epsilon \mid & \epsilon & = \mathsf{bin}(1,2), & 12 & = 11, & 112 & = \mathsf{lf}(6) \\ & 1 & = \mathsf{bin}(11,12), & 111 & = \epsilon, & 2 & = \mathsf{lf}(9) \\ & 11 & = \mathsf{bin}(111,112)\}. & & & & \end{array}$$

Using domain-theoretic semantics, we can obtain its expansion as an infinite term

$$\mathsf{bin}(\mathsf{bin}(\mathsf{bin}(\cdots, \mathsf{lf}(6)),\ \ \mathsf{bin}(\cdots, \mathsf{lf}(6))),\ \ \mathsf{lf}(9)) \tag{8.1}$$

where each "$\cdots$" is actually an infinitary long that repeats the whole term. This is regarded as an expansion of the structure in which each pointer node "$\swarrow p \uparrow i$" is connected directly to the referred node.

Defining this idea in a lazy functional language based on domain-theoretic semantics such as Haskell yields another interesting representation related to the use of internal pointer structures. Let's consider this in Haskell. Let the type HTree be a lazy datatype of trees defined by

$$t ::= \mathsf{lf}(k) \mid \mathsf{bin}(t_1, t_2)$$

(but here, for simplicity, we retain mathematical notation rather than Haskell). Consequently, we define the translation function $\mathsf{trans} : \mathsf{EGraph} \longrightarrow \mathsf{HTree}$ from equational term graphs to HTree by

$$\mathsf{trans}(\{y_1 \mid y_1 = r_1, \ldots, y_n = r_n\}) \ = \ \textbf{let}\ (\vec{x}) = (\vec{r})[\vec{y} \mapsto \vec{x}]\ \textbf{in}\ x_1 \tag{8.2}$$

where vectors denote sequences, and $[- \mapsto -]$ a substitution function of variables (written in Haskell). At the level of Haskell, this gives a translation into internal pointer structures in the heap memory of an implementation, because a **let**-expression (which is theoretically **letrec**) generates a pointer structure as presented in Fig. 5 because of the graph reduction mechanism of Haskell. Printing it will generate an infinite term as Eq. (8.1). In this way, starting from T via equational term graphs, our cyclic sharing terms can be used as "blueprints" of pointer structures in the memory.

A problem in the pointer structures is lack of structural induction. Exactly how it is possible to compose and decompose the pointer structures cleanly at the level of Haskell programming remains unclear. Therefore, this approach was thought to be somewhat insufficient for our goals, but this approach is nevertheless efficient and interesting.

9. Conclusion

We have given an initial algebra characterisation of cyclic sharing structures and derived inductive datatypes, structural recursion, and structural induction on them. We have also associated them with equational term graphs in the initial algebra framework. Hence we have shown that various ordinary semantics of cyclic sharing structures are applied equally to them.

From a programming perspective, practicality of our datatype of cyclic sharing structures must still be investigated. A possible direction of future work is to seriously use a dependently-typed programming language such as Coq and Agda for programming with cyclic sharing structures as an extension of this work.

Acknowledgement

The basis for this work, which was motivated by a question of Zhenjiang Hu, was done while the author visited IPL, University of Tokyo during April 2007 - March 2008. I express my sincere gratitude to Masato Takeichi and members of IPL for the opportunity to stay in a stimulating and pleasant research environment. I am also grateful to Varmo Vene and Tarmo Uustalu for discussions about datatypes in Haskell and reading early draft, and Shin-Cheng Mu for a discussion about dependent types in Agda. This work is supported by the JSPS Grant-in-Aid for Scientific Research (19700006) and NII collaboration research grant.

References

[AAMV03] P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, 300(1-3):1–45, 2003.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.

[Bro05] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proc. of TABLEAUX'05*, LNAI 3702, pages 78–92, 2005.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *Parallel Architechtures and Languages Europe*, LNCS 259, pages 141–158, 1987.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CGZ05] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Proc. of POPL'05*, pages 271–282, 2005.

[CKV74] B. Courcelle, G. Kahn, and J. Vuillemin. Algorithmes d'equivalence et de reduction a des expressions minimales dans une classe d'equations recursives simples. In *Proc. of ICALP*, pages 200–213, 1974.

[Dyb94] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

[Fio02] M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proc. of PPDP'02*, pages 26–37, 2002.

[Fio08] M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of LICS'08*, pages 57–68, 2008.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Annual Symposium on Logic in Computer Science*, pages 193–202, 1999.

[GHUV06] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proc. of TFP'06*, pages 173–188, 2006.

[GJ07] N. Ghani and P. Johann. Initial algebra semantics is enough! In *Proc. of TLCA'07*, LNCS 4583, pages 207–222, 2007.

[GTW76] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T. J. Watson Research Center, 1976. Reprinted in: Yeh, R. (ed.): Current trends in programming methodology IV. Prentic-Hall 1987, pp. 80-149.

[GUH06] N. Ghani, T. Uustalu, and M. Hamana. Explicit substitutions and higher-order syntax. *Higher-Order and Symbolic Computation*, 19(2/3):263–282, 2006.

[Ham04] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Proc. of APLAS'04*, LNCS 3302, pages 348–363, 2004.

[Ham05] M. Hamana. Universal algebra for termination of higher-order rewriting. In *Proc. of RTA'05*, LNCS 3467, pages 135–149, 2005.

[Ham09] M. Hamana. Initial algebra semantics for cyclic sharing structures. In *Proc. of TLCA'09*, LNCS 5608, pages 127–141, 2009.

[Has97] M. Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. PhD thesis, University of Edinburgh, 1997.

[Has02] R. Hasegawa. Two applications of analytic functors. *Theor. Comput. Sci.*, 272(1-2):113–175, 2002.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory.* Studies in Logic and the Foundations of Mathematics 141. North Holland, Elsevier, 1999.

[JG08] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proc. of POPL'08*, pages 297–308, 2008.

[McB02] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(4&5):375–392, 2002.

[MS03] M. Miculan and I. Scagnetto. A framework for typed HOAS and semantics. In *Proc. of PPDP'03*, pages 184–194. ACM Press, 2003.

[Nor07] U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[OS08] N. Oury and W. Swierstra. The power of Pi. In *Proc. of ICFP'08*, pages 39–50, 2008.

[PVWW06] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP '06*, pages 50–61, 2006.

[Rob02] E. Robinson. Variations on algebra: monadicity and generalisations of equational theories. *Formal Aspects of Computing*, 13(3-5):308–326, 2002.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput*, 11(4):763–783, 1982.

[SP00] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proc. of LICS'00*, pages 30–41, 2000.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.